RC24967 (W1001-053) January 14, 2010
Computer Science

IBM Research Report

A Model for Fusion and Code Motion in an Automatic Parallelizing Compiler

Uday Kumar Bondhugula, Oktay Günlük,
Sanjeeb Dash, Lakshminarayanan Renganarayana
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

A Model for Fusion and Code Motion in an Automatic Parallelizing Compiler

Abstract

Loop fusion has been studied extensively, but in a manner isolated from other transformations. This was mainly due to the lack of a powerful intermediate representation for application of compositions of high-level transformations. Fusion presents strong interactions with parallelism and locality. Currently, there exist no models to determine good fusion structures integrated with all components of an auto-parallelizing compiler. This is also one of the reasons why all the benefits of optimization and automatic parallelization of long sequences of loop nests spanning hundreds of lines of code have never been explored.

We present a fusion model in an integrated auto-parallelization framework that simultaneously optimizes for hardware prefetch stream buffer utilization, locality, and parallelism. Characterizing the legal space of fusion structures in the polyhedral compiler framework is not difficult. However, incorporating useful optimization criteria into such a legal space to pick good fusion structures is very hard. The model we propose captures utilization of hardware prefetch streams, loss of parallelism, as well as constraints imposed by privatization and code expansion into a single convex optimization space. The model scales very well to program sections spanning hundreds of lines of code. It has been implemented into the polyhedral pass of the IBM XL optimizing compiler. Experimental results demonstrate its effectiveness in finding good fusion structures for codes including SPEC benchmarks and large applications. An improvement ranging from 5% to nearly a factor of 2.75× is obtained over the current production compiler optimizer on these benchmarks.

1 Introduction

Currently, the trend in microarchitecture design is towards more processing elements on a single chip. Until the early 2000s, increasing clock frequencies boosted software performance without additional programming effort, or improvements in compiler and language design. This is well-known to no longer be true. There is a greater need for effective auto-parallelization in compilers.

Loop nest optimization has been studied extensively for several decades. Before the 2000s, most works were restricted to very narrow domains, typically perfect nests, with a single nest optimized at a time. There was less focus on composing a long sequence of transformations, as is necessary in practice to generate high performance code. As a result, analyses were restricted to small sections, and a very small subset of transformations were explored. Synergistic effects between various transformations were also lost. For example, it is known that parallelization on multicores does not provide good scaling unless single thread locality is simultaneously improved. Similarly, without privatization of data, parallelization often cannot be achieved. Significant improvement in performance can come from movement and fusion of code in large sections with several hundreds of lines of code. Hence, without a framework that can represent large sections of code to perform a number of high-level transformations in an integrated manner, the full benefits of automatic parallelization can neither be seen nor any conclusions be made about its effectiveness.

In the past eight years or so, the polyhedral model [2, 9] has emerged as a robust intermediate representation to extract from within a compiler for application of nearly all high-level optimizations. Contrary to common belief, the framework is not restricted to code with affine data accesses or static control flow. With recent advances, nearly any section of code within a single procedure, that only makes pure function calls if any, can be handled. This includes code with dynamic control flow [3], indirect accesses, and while loops. Conservative assumptions are made when necessary and to the extent needed. With recent advances in automatic transformation [4, 5], it is possible to find good sequences of transformations for coarse-grained outer parallelism, pipelined parallelism, cache and

register tiling for locality, and generate very efficient code [19, 16]. However, the lack of a fusion model is a roadblock in optimizing large applications. In particular, choosing the right outermost fusion structure has a big impact on how well the code can be transformed. It is this problem that we address here in a manner integrated with all other optimization components of an auto-parallelizer.

The choice of a fusion structure has trade-offs with parallelism and locality. Fusing maximally can hinder parallelization as it would typically increase the number of dependences satisfied on fused loops. Secondly, processors provide a limited number of hardware prefetch stream buffers, and excessive fusion may not utilize hardware prefetching well. One would like to use a number of prefetch streams that is as close as possible to the available number. On the other hand, distributing maximally maximizes parallelization opportunities, but leads to loss of locality as well as possibly reduced utilization of prefetching.

To the best of our knowledge, the fusion model we propose in this paper is the first one in a generic automatic transformation framework. It is also the first to capture prefetch stream buffer utilization as well as parallelization and privatization issues. The framework has been implemented in the polyhedral pass of the IBM XL compiler for C/C++/Fortran. Past works [9] have shown feasibility of polyhedral techniques on SPEC benchmarks through manual application of transformations and for sequential compilation. This work is the first to demonstrate its fully automatic application to complete benchmarks along with parallelization.

2 Background and Notation

The program or a section of it that has been extracted for optimization is a set of statements, S_1, S_2, $\ldots$, S_n. Loops surrounding each statement form its computational domain, and the iterations of the loops can be represented by integer points in a convex polyhedron.

Data Dependence Graph. The Data Dependence Graph (DDG) $G = (V, E)$ is a directed multi-graph with each vertex representing a statement. An edge, $e \in E$, from node S_i to S_j represents a

dependence with the source and target conflicting accesses in S_i and S_j respectively. The conditions on when the dependence exists are captured by the dependence polyhedron, $P_{e^{S_i \rightarrow S_j}}$, that relates the dependent source and target iterations through a system of linear constraints. The DDG and strongly-connected components (SCC) of the DDG are important entities when considering fusion.

Lemma 1 *All statements belonging to a strongly-connected component of the data dependence graph have a common surrounding loop, i.e., they cannot be distributed [11].*

Transformations: loop hyperplanes and partitionings: A transformation for a program is a statement-wise multi-dimensional affine function. Let $\vec{i}_S$ be an iteration in the domain of a statement S. Each dimension or *level* of a statement-wise transformation can be represented as follows:

$$\phi_S(\vec{i}) \quad = \quad \begin{pmatrix} c_S^1 \; c_S^2 \; \dots \; c_S^{m_S} \end{pmatrix} \begin{pmatrix} \vec{i}_S \end{pmatrix} + c_S^0 \tag{1}$$

Let ϕ refer to $\{\phi_{S_1}, \phi_{S_2}, \dots, \phi_{S_n}\}$. ϕ is a *fusion partitioning* iff $c_S^1 = c_S^2 = \dots = c_S^{m_S} = 0, \forall S$. In this case, ϕ partitions the set of statements in a particular order, c_S^0 being the partition number for S. ϕ is a *loop hyperplane* iff $\sum_{i=1}^{m_S} c_S^i \geq 1$.

Loop hyperplanes specify a fused loop while a partition serves the purpose of distributing statements. Hence, some rows of the multidimensional affine transformation represent loops while the rest are partitions interspersing them. In the literature, there exist techniques to find loop hyperplanes that maximize tiling opportunity in order to simultaneously optimize for coarse-grained parallelism and locality. The focus of this paper is on fusion partitionings, but we will need the following condition. For a loop hyperplane to not violate an unsatisfied dependence edge $e \in E$, the following must hold true.

$$\phi_{S_j}\left(\vec{t}\right) - \phi_{S_i}\left(\vec{s}\right) \quad \geq \quad 0, \quad \left\langle \vec{s}, \vec{t} \right\rangle \in \mathcal{P}_{e^{S_i \rightarrow S_j}} \tag{2}$$

The above constraint can be cast into a set of linear inequalities just involving ϕ's coefficients with techniques known in the polyhedral literature [4]. In the rest of this paper, we refer to the constant $c_{S_i}^0$ of a particular statement S_i as c_i whenever ϕ represents a partitioning as in the context

of fusion.

3 Fusion Model

In this section, we describe the optimization criteria and cost models to pick a single good fusion structure from the space of all possible legal choices.

3.1 Legal space of fusion structures

For any two statements, S_i and S_j, one of the following can be concluded from data dependences:

(i) Strong fuse: $c_i = c_j$

(ii) Weak fuse (forward): $c_i \leq c_j$

(iii) Weak fuse (backward): $c_j \leq c_i$

(iv) Strong distribute (forward): $c_i \leq c_j + 1$

(v) Strong distribute (backward): $c_j \leq c_i + 1$

(vi) Unrestricted: c_i, c_j unconstrained with respect to each other

Case (i) applies when S_i and S_j are in the same SCC; by Lemma 1 they must be fused together. Case (vi) applies if S_i and S_j are unconnected in the DDG. Cases (ii), (iii), (iv), and (v) apply when S_i and S_j are weakly connected in the dependence graph, i.e., either when there exists a path from S_i to S_j, or from S_j to S_i: the former would lead to either (ii) or (iv), while the latter will lead to (iii) or (v). However, distinguishing between (ii) and (iv), or between (iii) and (v), requires more analysis. Certain dependences do not permit fusion. This analysis can be done using constraint (2), but by restricting it to dependences that have the concerned statements as their source and target. Consider the following condition:

$$\phi_{S_y}\left(\vec{t}\right) - \phi_{S_x}\left(\vec{s}\right) \quad \geq \quad 0, \quad \left\langle \vec{s}, \vec{t} \right\rangle \in \mathcal{P}_{e^{S_x \rightarrow S_y}}, \quad \forall e \in E \text{ such that } S_x, S_y \in \{S_i, S_j\} \tag{3}$$

If valid ϕ loop hyperplane solutions cannot be found for the above, we infer case (iv) as opposed to (ii), or (v) as opposed to (iii). Also, if two statements are to be distributed, all statements in the SCC comprising the first one need to be distributed away from all statements in the SCC comprising the second; this is obviously implied by case (i). Hence, the above analysis can be done on an SCC pair-wise basis, as opposed to statement pair-wise. In fact, all six conditions above can be written on an SCC-basis. We still choose to present on a statement-wise basis for better clarity. Thus, a system of linear equalities and inequalities can be built by deducing one of the six cases for every (S_i, S_j). This convex space in the c_is is the set of all legal fusion choices, and we denote this by $\mathcal{L}$.

Consistency: Pouchet et al. [14] provide detailed properties related to transitivity while constructing the set of all legal distinct fusion structures for the purpose of iterative search. Fusion is not transitive when permutation and skewing interfere. In such cases, a legal solution to $\mathcal{L}$ will need further distribution in order to represent a partitioning. In practice, we find that such cases arise in an very small fraction of codes, and we thus stay with the above simple construction. A fallback solution is easily obtained when such interference exists. If need be, a complete and consistent legal space can be constructed at a higher cost using [14]. The model proposed in this paper can be used in such a context as well.

3.2 Optimizing for prefetch stream buffers

The problem now is to augment the legal space of fusion structures with cost models for optimization criteria. One of the desired goals is to ensure that no fused nest consumes more than the available number of hardware prefetch streams. Other goals are to make sure that fusion is conducive for parallelization as well as respects constraints imposed by privatization of data. Optimizing for utilization of prefetch stream buffers is the most challenging of these, and we address it in this section.

Most modern hardware supports prefetching. Data required by accesses to contiguous locations of the memory can be prefetched instead of being supplied on-demand. Hence, an array access $a[i][j]$

with i, j being the loop iterators in that order, can make use a hardware prefetch stream buffer while accessing along a row. Compilers can insert code to initiate prefetch streams for candidate accesses. Constant strided accesses are also candidates for prefetching. Fusion of statements that access the same data would allow data reuse as well as reuse or sharing of a prefetch stream. Hence, the number of prefetch streams that will be used is not simply the sum of those used individually by each of the statements in the same partition.

It is difficult to determine a priori the requirements of each statement since the inner loop structure of the fused program itself would not be known while determining the outermost distribution. We go with a worst-case estimation, i.e., by assuming that for the fusion structure that will be chosen, the rest of the transformation framework that finds inner loop hyperplanes would maximize spatial reuse thereby increasing the number of accesses that would require prefetch streams.

From the linear constraints in Section 3.1, the set of legal fusion/distributions is a convex space in the c_is, i.e., partition numbers. A challenge in constructing an objective is the hardness of encapsulating optimization criteria involving just c_is, i.e., partition numbers the statements belong to. This problem is addressed by introducing a set of binary decision variables that provide greater power in capturing statements comprising a partition. Let x_{ij} be a binary decision variable such that

$$x_{ij} = \begin{cases} 1, & \text{if } S_i \text{ is in partition } j \\ 0, & \text{if } S_i \text{ is not in partition } j \end{cases}$$

Due to Lemma 1, the number of partitions can never be greater than the number of SCCs. Let the number of partitions be N. N is a variable and will only be known when a solution is found.

Let T be the following constant table, i.e., its values are known at compile-time: rows correspond to statements and columns correspond to prefetch stream requirements of statements. Let M be the number of columns of the table. In some cases, a single data space may need multiple prefetch streams. For example, in C code, accesses $a[i][j]$, $a[i+1][j]$, with loop iterators, i and j would require separate prefetch streams, and so they will have distinct columns in T. M is thus roughly of the

Stmts/Data spaces	D_1	D_2	$\dots$	D_M
S_1	1	1	$\dots$	0
S_2	0	1	$\dots$	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
S_n	1	0	$\dots$	1

Table 1. Statement / prefetch stream requirement table: T

order of the number of distinct data spaces (arrays/matrices) accessed across all statements. Based on prefetch stream requirements of each statement, one can populate T as follows. Table 1 shows a sample table.

$$T_{ik} = \begin{cases} 1, & \text{if } S_i \text{ requires a stream for array } k \\ 0, & \text{if } S_i \text{ does not require a stream for array } k \end{cases}$$

Let z_{kj} be a binary decision variable such that

$$z_{kj} = \begin{cases} 1, & \text{if a stream is needed for array } k \text{ in partition } j \\ 0, & \text{if no stream is needed for array } k \text{ in partition } j \end{cases}$$

With these variables, it is now possible to add constraints that capture prefetch stream requirement. We can express the relation between the z and x variables as follows:

$$z_{kj} = \bigvee_{i|T_{ik}=1} x_{ij} \tag{4}$$

Since $z_{kj} \in \{0,1\}$, $x_{ij} \in \{0,1\}$, the above in turn can be written as:

$$z_{kj} \geq x_{ij}, \quad \forall i \text{ such that } T_{ik} = 1 \tag{5}$$

One can express the partition number as:

$$c_i = \sum_{j=1}^{N} j * x_{ij} \tag{6}$$

Now, the number of prefetch streams for partition j is just $\sum_k z_{kj}$, and can be enforced, for a processor with 7 prefetch streams, as:

$$\sum_{k=1}^{M} z_{kj} \leq 7, \quad 1 \leq j \leq N \tag{7}$$

The number on the RHS is known at compile-time based on the target processor and on whether the compiler has been asked to auto-parallelize. Streams are shared equally among all threads on a chip. Seven to twelve streams are common for example.

The objective that now really fits well is one that minimizes the number of partitions subject to above constraints. We now combine all the above constraints and specify the Integer Programming formulation. All constraints in $\mathcal{L}$, as defined in Section 3.1, fall into one of the following three sets.

$$P_i^{=} = \{k \mid c_i - c_k = 0\} \tag{8}$$

$$P_i^{+} = \{k \mid c_i - c_k \geq 0\} \tag{9}$$

$$P_i^{++} = \{k \mid c_i - c_k \geq 1\} \tag{10}$$

Therefore, the minimum number of partitions can be obtained by solving the following integer program (IP):

$$\text{minimize } c_{max} \tag{11}$$

$$\sum_{j=1}^{N} x_{ij} = 1 \qquad \forall i \tag{12}$$

$$c_i = \sum_{j=1}^{N} j * x_{ij} \qquad \forall i \tag{13}$$

$$c_i = c_k \qquad \forall i, k \text{ such that } k \in P_i^{=} \tag{14}$$

$$c_i \geq c_k \qquad \forall i, k \text{ such that } k \in P_i^{+} \tag{15}$$

$$c_i \geq c_k + 1 \qquad \forall i, k \text{ such that } k \in P_i^{++} \tag{16}$$

$$c_{max} \geq c_i \qquad \forall i \tag{17}$$

$$z_{kj} \geq x_{ij}, \qquad \forall i, j, k \text{ such that } T_{ik} = 1 \tag{18}$$

$$\sum_{k=1}^{M} z_{kj} \leq 7 \qquad \forall j \tag{19}$$

$$x_{ij} \in \{0, 1\}, \quad z_{kj} \in \{0, 1\} \tag{20}$$

where the number of variables and constraints linearly depend on N. Clearly, from a computational point of view, it is desirable to obtain a formulation with fewer variables and constraints and therefore it is desirable to choose the number N as small as possible. Notice that N can be chosen to be as small as c_{max} which is not known in advance. Instead, we use heuristics to obtain an upper bound on c_{max} and use the resulting bound as the number N.

3.3 A greedy heuristic

The first heuristic we use to find an upper bound on c_{max} is a very fast and simple greedy heuristic. In this heuristic, we construct partitions sequentially by simply picking the first feasible statement and assigning it to the current partition. A statement is considered feasible if it does not have any fusion/distribution restrictions (modulo current partial solution) and if its data streams can be prefetched without violating the limit. Statements that must be fused with the current statement in consideration (i.e. the set $P_i^=$ if the current statement is i) are assigned to the same partition simultaneously. If no more statements can be assigned to the current batch, we simply start a new batch.

Any solution found by this simple procedure is clearly an upper bound on c_{max} and therefore can be used as N in the formulation. But if the above procedure fails to find a solution (due to fusion/distribution restrictions), this does not necessarily mean that no feasible solution exists. More precisely, if the precedence graph has directed cycles formed by pairs of statements that can be fused, for example, let $c_i \geq c_j$, $c_j \geq c_k$, $c_k \geq c_i$, then the above procedure will get stuck as all statements in this example, namely, i, j and k can be assigned to a batch only after their predecessors are already assigned. To solve this problem, one needs to identify the strongly connected components in the precedence graph to identify collections of statements that need to be fused. Consequently, if the heuristic fails to find a solution, we perform an extra step and find all directed cycles to identify the implied fusion requirements. With this extra information, the heuristic can now find a feasible solution if one exists.

3.4 Dealing with infeasibility

There are two possible reasons for the IP in Section 3.2 to be infeasible. The first reason is that any feasible solution satisfying the fusion/distribution restrictions requires more hardware prefetch streams than specified by constraint (19). In this case, one needs to increase the number of prefetch streams to make the problem feasible. If we define the minimum "spill" to be the minimum increase in this number, what we do in this case is to find a solution which first minimizes the spill and then minimizes the number of partitions with that spill. We do this in two steps.

To find the minimum "spill" we simply use binary search on the size of the spill and check if the resulting relaxed IP is feasible using the heuristic described above. Once we find the minimum "spill", we increase the right hand side of constraint (19) in the IP by this number. Now that the relaxed problem is feasible, the second step is to first apply the heuristic to find the number N and then solve the IP to obtain the minimum number of partitions. We note that it is also possible to find the minimum spill using integer programming by formulating a different model.

Note that IP can also be infeasible due to inconsistent fusion/distribution restrictions. More precisely, there might be a collection of distribution restrictions forming a directed cycle. For example, if one has the following restrictions $c_i \geq c_j$, $c_j \geq c_k$, $c_k \geq c_i + 1$, then there is no feasible solution to the problem. To identify this type of infeasibility, we check if having as many prefetch streams as there are data streams in the problem has a feasible solution. If there is, we can be sure that by increasing the spill, we can obtain a feasible problem. If not, we conclude that the infeasibility is due to inconsistent fusion/distribution restrictions. Clearly, this should never be the case.

3.5 Solving the IP

We use the Coin-Cbc [6] code to solve the integer program formulated with N defined as the upper bound found by the heuristic *minus* 1. If the resulting IP is feasible, we obtain a solution that is strictly better than the solution found by the heuristic. If, on the other hand, the IP is infeasible,

we conclude that the solution found by the heuristic is already optimal. This simple idea in practice speeds up the IP solution time noticeably.

3.6 Interaction with parallelization

Modeling fusion while also enabling good parallelization is known to be notoriously hard. In this section, we propose an approach that is computationally efficient, but makes some practical trade-offs. It can be described in terms of additional constraints that can be added to $\mathcal{L}$, the legal fusion space constructed in Section 3.1.

```
/* Center the column vectors. */
for (i = 1; i <= n; i++)
 for (j = 1; j <= m; j++)
  data[i][j] -= mean[j];

/* Calculate m*m covariance matrix. */
for (j1 = 1; j1 <= m; j1++) {
 for (j2 = j1; j2 <= m; j2++) {
   symmat[j1][j2] = 0.0;
   for (i = 1; i <= n; i++) {
    symmat[j1][j2] += data[i][j1]*data[i][j2];
   }
   symmat[j2][j1] = symmat[j1][j2];
 }
}
```

(a) Covcol

```
for (i=0; i<N; i++)
 for (j=0; j<N; j++)
  B[i][j] = A[i][j]+u1[i]*v1[j]+u2[i]*v2[j]; //S1

for (i=0; i<N; i++)
 for (j=0; j<N; j++)
  x[i] = x[i] + beta* B[j][i]*y[j]; //S2

for (i=0; i<N; i++)
 x[i] = x[i] + z[i]; //S3

for (i=0; i<N; i++)
 for (j=0; j<N; j++)
  w[i] = w[i] + alpha* B[i][j]*x[j]; //S4
```

(b) GEMVER

Figure 1. Fusion parallelization issues

Examples: Note the way variable *data* is used in the second nest in Figure 1(a). Fusing the first with the second will result in a nest that has no outer parallelism. No permutation or any other affine transformation of the fused nest will yield an outer loop that can be parallelized. We show we are able to capture such interaction in the fusion model directly.

An SCC C_i has outer parallelism iff there exists loop hyperplane ϕ satisfying

$$\phi_{S_y}\left(\vec{t}\right) - \phi_{S_x}\left(\vec{s}\right) = 0, \quad \left\langle \vec{s}, \vec{t} \right\rangle \in \mathcal{P}_{e^{S_x \to S_y}}, \quad \forall e \in E \text{ such that } S_x, S_y \in C_i \tag{21}$$

Outer parallelism is preserved by fusion of SCC C_i and SCC C_j iff

$$\phi_{S_y}\left(\vec{t}\right) - \phi_{S_x}\left(\vec{s}\right) \quad = \quad 0, \quad \left\langle \vec{s}, \vec{t} \right\rangle \in \mathcal{P}_{e^{S_x \to S_y}}, \quad \forall e \in E \text{ such that } S_x, S_y \in \{C_i \cup C_j\} \tag{22}$$

1. Check each SCC individually for outer parallelism by checking (21)

2. For every pair of SCCs in the dependence graph that are (weakly) connected and with at least one of them parallel, check for loss of outer parallelism using (22). If parallelism is lost, add a constraint to $\mathcal{L}$ for strong distribution of this pair of SCCs.

Due to Step 2, any loss in outer parallelism that is by itself due to fusion of the two SCCs is detected. Except for the caveat mentioned in Section 3.1, the resulting constraints completely capture any loss in parallelism due to fusion choice. These constraints by themselves do not add any inconsistency since they merely convert "weak fuse" (Section 3.1) to "strong distribute". For the code in Fig. 1(b), the above leads to a constraint that separates S_3 and S_4, i.e., $c_4 - c_3 \geq 1$. The partitioning we finally end up with is $\{S_1, S_2\}$, $\{S_3\}$, $\{S_4\}$ – with all four fused nests exhibiting outer parallelism as well as improved reuse within the first partition.

3.7 Interaction with privatization

When developing a sequential application, a programmer usually reuses the same storage repeatedly in every loop iteration. This introduces unnecessary false dependences that can be eliminated by privatizing or expanding the variable. Such variables or temporaries can also be generated by other passes of the compiler itself. Two choices arise when handling these expanded variables after transformation.

1. The variable can be left expanded and a buffer be created for it

2. If transformations permit, the variable can be marked as local to a transformed loop. If the loop is parallelized, it will be made OpenMP private.

If Option 1 is chosen and loop trip counts are not known at compile time or when they are large, one could either end up with large buffers or buffers of unknown size that have to be allocated dynamically. This is a scenario we would always like to avoid in generated compiler code: the allocated buffer may cause cache pollution. With Option 2, one would end up with exactly as many copies as the number

```
for (r=0; r< N; r++) {
  for (q=0; q< N; q++) {
        for (p=0; p< N; p++) {
          sum[p] = 0.0; // S1
          for ( s = 0; s< N; s++) {
                sum[p] = sum[p] + A[r][q][s]*C4[s][p];   // S2
          }
        }
        for (p=0; p< N; p++) {
          A[r][q][p] = sum[p]; // S3
        }
  }
}
```

Figure 2. Fusion privatization issues

of processors if the loop is parallelized. However, for this to be achieved, transformations should not distribute statements accessing a privatized variable along its expanded dimensions. Distribution would necessitate expansion. To prevent such distribution, additional constraints can be added to $\mathcal{L}$. These constraints keep necessary statements together on all levels at which the variable had been privatized. For the example below in Figure 2, we end up adding $c_1 - c_2 = 0$, $c_2 - c_3 = 0$. In some cases, though tiling can necessitate creation of buffers for these expanded variables, they are of a fixed size proportional to tile sizes and do not pose a problem.

4 Putting it all together

Once $\mathcal{L}$ is constructed as per Section 3.1, constraints imposed by privatization (Section 3.7) are added followed by constraints for parallelization (Section 3.7). We also have a heuristic to prevent fusion that would result in code expansion. Code expansion can occur due to fusion of statements with iteration spaces bounded by different symbols. Again, constraints to distribute them can be added.

Conflicts in precedence constraints: Since constraints enforced to prevent loss of parallelism would separate particular SCCs while privatization would try to keep a set of statements together, there could be an inherent conflict between the two. This is easy to check by verifying feasibility of the space before parallelization constraints are added: if the resulting space becomes infeasible, parallelization constraints are discarded in favor of privatization constraints. A conflict due to code expansion constraints are dealt with similarly.

Once the above space is built, it is provided to the LP/IP-based model described in Section 3.2. Since the solution provided by it minimizes the number of partitions, locality optimization is automatically achieved subject to those constraints. Improvement in locality due to fusion is further exploited by both tiling and register tiling of the fused nests. The fusion model is used to first determine the outermost fusion structure, and maximal fusion is employed for all subsequent inner levels.

Compilation time The fusion model implementation runs very fast and has negligible impact on the overall compilation time. For the codes evaluated in the next section, which we believe are substantial, in no case does the fusion model take more than one second to provide a solution. In any case, we also use a time bound on the IP formulation, i.e., if it were to take more than a certain number of seconds, the best of the heuristic solution and the IP solution found till then, if any, is taken not worrying about provable optimality.

5 Experimental evaluation

Setup and comparison The machine used for experiments is an IBM Power5 4-way SMP system, with each Power5 processor being a 1.65 GHz dual-core with a 32 KB L1 D cache, a 1.9 MB shared L2 cache, and a 36 MB off-chip L3 cache. SMT functionality was not used. The IBM XL compiler v11.1 for C/C++ and v13.1 for Fortran was used with optimization flags: `-O3 -qhot -qtune=pwr5 -qarch=pwr5` with `-qsmp -qthreaded` added to enable auto-parallelization. At `-O3 -qhot`

-qsmp, the existing optimizer in XL performs significant loop and high-level optimizations of its own, including a heuristic to perform fusion based on similar optimization criteria; it is referred to as *xl-seq* and *xl-smp*, depending on sequential or parallelized code. The polyhedral pass is enabled by providing additional flags. *xl-poly-smartfuse* refers to our new fusion model. We compare with simple fusion choices of "maximal fusion" as well as "no fusion", referred to as *xl-poly-maxfuse* and *xl-poly-nofuse* in the graphs respectively. A solution that minimizes the number of partitions at each level is a (greedy) maximal fusion solution. Completely distributing all strongly-connected components is also a valid solution and this would lead to no fusion. Both *maxfuse* and *nofuse*, like *smartfuse*, get all benefits of the polyhedral pass. Results in all cases show the combined benefit from complementary transformations which include all affine transformations, cache tiling, and register tiling. The polyhedral pass of the compiler incorporates state-of-the-art techniques for these orthogonal components – to find loop hyperplanes [4, 5], to perform register tiling [17, 16] with efficient code generation.

Benchmarks We consider benchmarks that are expected to be sensitive to fusion. Gemver is a linear algebra routine used in householder transformations, in matrix bidiagonalization and tridiagonalization. ls3df-cg is the Conjugate Gradient routine from ls3df, a program used for electronic structure calculations. covcol and corcol are from the Principal Component Analysis (PCA) benchmark suite. eon is a SPEC2000INT benchmark while wupwise and facerec are from SPEC2000FP. hmmer, milc, astar, and cactus are from SPEC2006, that nearly all major hardware vendors publish results for. Advect3D is an weather modeling application also reported in [15]. Parallelization results presented are on up to four cores, except in some cases where we see a different trend with more cores.

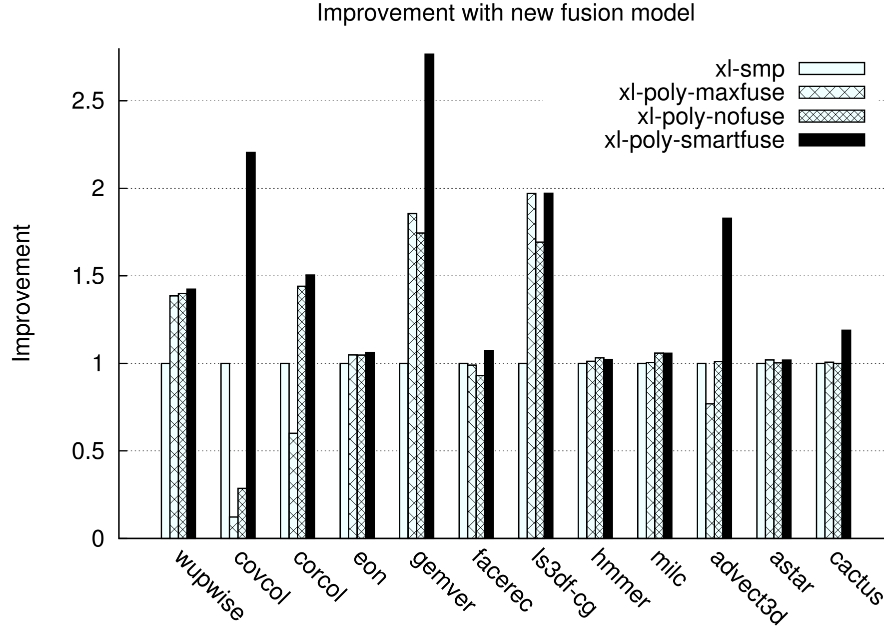
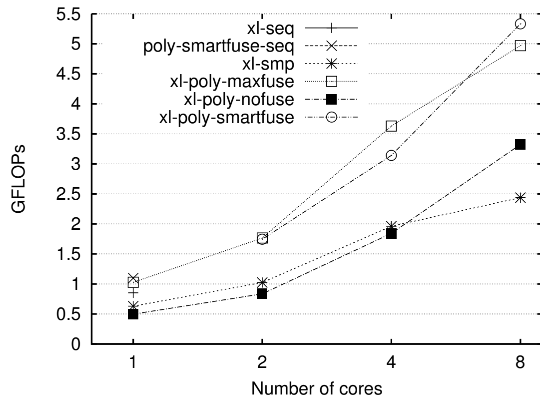


Figure 3. Performance improvement for all benchmarks

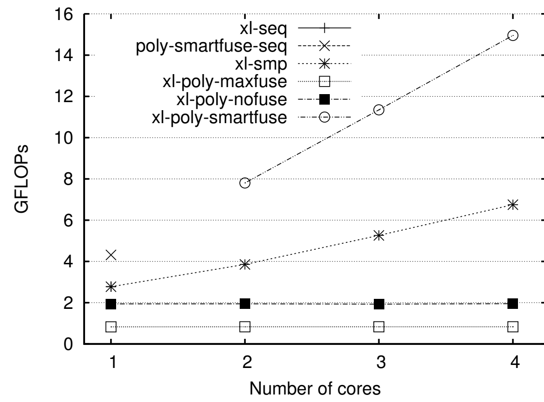
5.1 Analysis

Fig. 3 shows a summary of improvement over the existing compiler while running all codes on 4 cores. The fusion model is able to find the best performing fusion structure in nearly all cases. Though in some cases *nofuse* or *maxfuse* itself might be a good solution, the model's ability to capture the one that is the best is particularly interesting and important to note. In several cases, *smartfuse* provides a structure that is significantly better than *nofuse* and *maxfuse*, being different from either of those. For the three SPEC2000 benchmarks – eon, facerec, and wupwise, improvements of 5%, 6%, and 30% respectively, are obtained over the existing compiler. For the SPEC2006 benchmarks – hmmer, milc, astar, and cactusADM, improvements of 2%, 5.5%, 2.5%, and 16% respectively, are obtained.

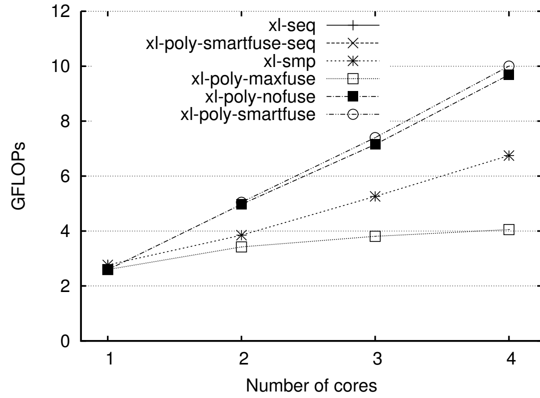
For covcol and corcol, *smartfuse* partitions statements into two groups even though all of them are fusable and distributable. For Gemver (Fig. 1(b)), the first two statements are fused with the first one being permuted: this improves locality, while the third and fourth statements are distributed. This solution preserves outer parallelism in all three fused nests, and the resulting code scales almost



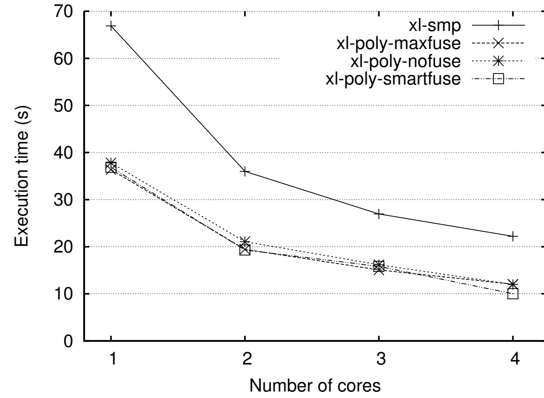
(a) GEMVER



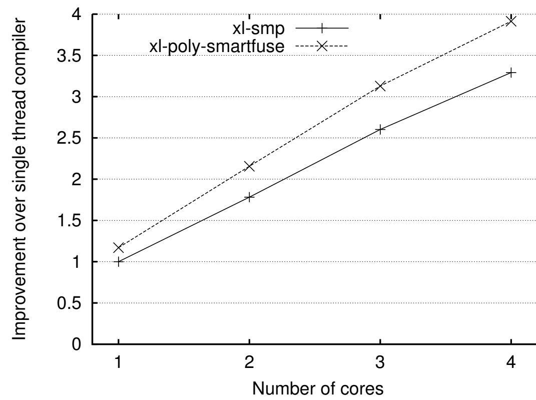
(b) Covcol



(c) Corcol



(d) ls3df-cg



(e) SPEC2006 CactusADM

Figure 4. Improvement with new fusion model

linearly with the number of cores. Maximal fusion destroys pure outer parallelism in this case, and only wavefront pipelined parallelization can be performed. *nofuse* leads to a complete loss of locality and hence performs poorly. The solution obtained by existing compiler optimizer (without the polyhedral pass) is closer to *nofuse*. Figures 4(a) and 4(b) show exact GFLOPs performances. *covcol* shows similar trends as well. *maxfuse* leads to a loss of outer parallelism. *nofuse* performance for covcol does not show any improvement with parallelization due to an unknown interaction with later compiler passes.

smartfuse provides a different fusion structure depending on whether the compiler is asked to auto-parallelize (Section 3.2). Hence, code generated for the sequential case with just locality optimization will typically use fewer partitions. In Figure 4(a) and Figure 4(b), note that the performance of poly-smartfuse-seq is significantly better. For ls3df-cg in Fig. 4(d), a solution closer to *maxfuse* is the best one, and there is a small improvement over *nofuse*. In this case, improvement with polyhedral techniques irrespective of the fusion structure is obtained as a result of an interchange on the key imperfect nest making a parallel loop outermost. All fusion structures get this benefit, and the benefits of inner loop fusion can be seen with *smartfuse* and *maxfuse* when running on fewer cores when the data set accessed by each thread is larger. Figure 4(e) shows *smartfuse* achieving nearly linear speedup for cactus-ADM. The improvement is seen with any number of threads due to improved single thread locality.

6 Related Work

Traditional works on loop fusion [10, 12, 18, 15] are restricted in their ability to find complex fusion structures. This is mainly due to the lack of a powerful representation for dependences and transformations. Hence, non-polyhedral approaches typically study fusion in an manner isolated with other transformations. Darte et al. [8, 7] study fusion with parallelization, but only in combination with shifting. Our work on the other hand enables fusion in the presence of all polyhedral transformations which include those that make tiling legal and enable better parallelization and locality optimization.

Megiddo and Sarkar [12] proposed a way to perform fusion for an existing parallel program by grouping components in a way that parallelism is not disturbed. Decoupling parallelization and fusion clearly misses several interesting solutions that would have been captured if both were treated together. Examples in Section 3.6 demonstrate this.

Lim et al.'s [11] affine partitioning algorithm treats each SCC independently. Hence, no choice of fusion structures across SCCs is considered. Bondhugula et al [4, 5]'s framework implemented in Pluto [13] performs transformations to enable coarse-grained parallelization and locality optimization through tiling and maximal fusion; it subsumes previous works based on affine partitioning [11, 1]. The tool can also perform complete distribution separating all SCCs if needed. However, no model exists to choose a good fusion structure based on any criteria.

Pouchet et al. [14] provide properties and techniques to build a convex space comprising the set of all legal and distinct fusion structures. The space is built in order to allow iterative empirical search through systematic enumeration. The space they construct can be used in place of the simpler one that we proposed in Section 3.1. Though it would incur higher cost, all optimization metrics developed in this paper would apply to it.

7 Conclusions

We presented a fusion model for an integrated auto-parallelization framework that simultaneously optimizes for hardware prefetch stream buffer utilization, locality, and parallelism. The proposed model also captures constraints imposed by privatization and code expansion. A single convex optimization space with an objective function is built incorporating all these. Results show that it scales very well to large applications including SPEC benchmarks. It has been fully implemented into the polyhedral pass of the IBM XL compiler's optimizer. Experimental results demonstrate its effectiveness in finding good fusion structures. An improvement ranging from 5% to nearly a factor of 2.75× is obtained over a highly tuned optimizing production compiler over a range of selected benchmarks on a multicore. To the best of our knowledge, this is the first fusion model to incorporate

such concrete optimization criteria as well as demonstrate improvement on large applications with the polyhedral framework.

Acknowledgments

This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under its PERCS program through contract HR0011-07-9-0002. We would like to acknowledge Louis-Noel Pouchet for past joint work on construction of legal fusion spaces.

References

[1] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loops. *International Journal of Parallel Programming*, 29(5), Oct. 2001.

[2] C. Bastoul. Extracting polyhedral representation from high level languages. Technical report, LRI, Paris-Sud University, 2008. Clan tool.

[3] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *Compiler Construction (CC)*, 2010.

[4] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *Compiler Construction (CC)*, Apr. 2008.

[5] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *ACM SIGPLAN PLDI*, June 2008.

[6] Computational Infrastructure for Operations Research. http://www.coin-or.org.

[7] A. Darte and G. Huard. Loop shifting for loop parallelization. Technical Report RR2000-22, ENS Lyon, May 2000.

[8] A. Darte, G.-A. Silber, and F. Vivien. Combining retiming and scheduling techniques for loop parallelization and loop tiling. *Parallel Processing Letters*, 7(4):379–392, 1997.

[9] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations. *IJPP*, 34(3):261–317, June 2006.

[10] K. Kennedy and K. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *LCPC*, pages 301–320, 1993.

[11] A. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24(3-4):445–475, 1998.

[12] N. Megiddo and V. Sarkar. Optimal weighted loop fusion for parallel programs. In *symposium on Parallel Algorithms and Architectures*, pages 282–291, 1997.

[13] PLUTO: A polyhedral automatic parallelizer and locality optimizer for multicores. http://pluto-compiler.sourceforge.net.

[14] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Hybrid iterative and model-driven optimization in the polyhedral model. Technical Report 6962, INRIA Research Report, June 2009.

[15] A. Qasem and K. Kennedy. Profitable loop fusion and tiling using model-driven empirical search. In *Proc. ACM International conference on Supercomputing*, pages 249–258, 2006.

[16] L. Renganarayana, U. Bondhugula, S. Derisavi, A. E. Eichenberger, and K. O'Brien. Compact multi-dimensional kernel extraction for register tiling. In *Supercomputing*, 2009.

[17] L. Renganarayana, D. Kim, S. Rajopadhye, and M. M. Strout. Parameterized tiled loops for free. In *ACM SIGPLAN PLDI*, pages 405–414, 2007.

[18] S. Singhai and K. McKinley. A Parameterized Loop Fusion Algorithm for Improving Parallelism and Cache Locality. *The Computer Journal*, 40(6):340–355, 1997.

[19] N. Vasilache, C. Bastoul, and A. Cohen. Polyhedral code generation in the real world. In *International conference on Compiler Construction (CC)*, pages 185–201, Mar. 2006.